Generic Programming in POOMA and PETE

James A. Crotinger, Julian Cummings, Scott Haney,
William Humphrey, Steve Karmesin, John Reynders,
Stephen Smith, and Timothy J. Williams

Los Alamos National Laboratory; Los Alamos, NM 87545

Abstract. POOMA is a C++ framework for developing portable scientific applications for serial and parallel computers using high-level physical abstractions. PETE is the expression template library used by POOMA. This paper discusses generic programming techniques that are used to achieve flexibility and high performance in POOMA and PETE. POOMA uses an engine class that factors the data representation out of its array classes. PETE's expression templates are used to build up and operate efficiently on expressions. PETE itself uses generic techniques to adapt to a variety of client-class interfaces, and to provide a powerful and flexible compile-time expression-tree traversal mechanism.

1 Introduction

POOMA (Parallel Object-Oriented Methods and Applications) is an object-oriented framework for developing scientific computing applications on platforms ranging from laptops to parallel supercomputers [1, 2]. POOMA includes C++ template classes representing high-level mathematical abstractions such as arrays, particles, and fields. POOMA objects can be used in data-parallel expressions, with the parallelism encapsulated in the underlying framework. Expression creation and manipulation facilities are provided by PETE, the Portable Expression Template Engine.

This paper discusses generic programming techniques used to achieve flexibility and high performance in POOMA II and in PETE. POOMA II, currently under development, is a redesign of POOMA intended to further increase expressiveness and performance. POOMA arrays (the "II" will henceforth be understood) delegate data allocation and element access to a new *engine* class, allowing the array class to provide a uniform interface, including array expression capability, for a variety of data formats. Using PETE, POOMA separates the representation of an expression from its evaluation, allowing POOMA to provide multiple expression evaluation mechanisms. The simplest mechanism inlines the entire evaluation in a manner similar to conventional expression-template array classes. Alternatively, an expression can be subdivided into expressions on subdomains of the arrays, and these sub-expressions can be evaluated independently by multiple threads.

PETE uses generic techniques to avoid assumptions about client-class interfaces and to provide a powerful and flexible expression tree traversal mechanism.

This paper is organized as follows. Section 2 discusses the engine abstraction. Section 3 gives a brief introduction to expression templates. Section 4 describes the use of PETE to add expression template capability to client classes, and the method by which PETE performs expression object manipulations, including evaluation. Section 5 concludes the paper with a discussion of POOMA's expression-engine, an engine that allows expressions to be used a arrays.

2 Arrays and Engines

Many scientific computing applications require data types with multidimensional array semantics, but with a variety of underlying data structures. Examples range from regular arrays having Fortran or C storage order, to banded or sparse matrices, to array-like objects that compute their elements directly from their indices. One could model these data types using an inheritance hierarchy. An abstract base-class would define the interface, and descendent classes would override virtual functions to deal properly with their internal data structures. Unfortunately, the cost of virtual function calls in the inner loop of an array expression is prohibitive. Compile-time techniques are required to satisfy the performance requirements of most scientific applications.

POOMA's `Array` class achieves the desired flexibility by factoring the data representation into a separate engine class. The `Array` provides the user interface and expression capability, while the engine provides data storage and mapping of indices to data elements.

All engines are specialization of an `Engine` template class:

```
template <int Dim, class T, class EngineTag> class Engine { };
```

For example, a *brick-engine*, which stores a contiguous block of data that is interpreted as a multi-dimensional array with Fortran storage-order, is declared:

```
class Brick {};
template <int Dim, class T> class Engine<Dim,T,Brick>;
```

Partial specialization on an engine-tag is used so that `Array` can provide reasonable default template parameters:

```
template <int Dim, class T=double, class EngineTag=Brick>
class Array;
```

The `Array` class delegates individual element access to the engine class, which will return the appropriate element in the most efficient manner possible.

Note that the array is not the container of the data, it is a user interface for manipulating the data. As a result, a "`const Array`" is not what one might think. The following is completely legal:

```
Array<1> a(10);
const Array<1> & ar = a;
ar(4) = 3.0;
```

The assignment is allowed because it does not modify the `Array` object. This is similar to the distinction between an STL container and an STL iterator. POOMA's `ConstArray` class is a read-only array class that is analogous to the STL `const_iterator`. `ConstArray` is also the base class for `Array`. Not only is this natural from an implementation standpoint (`Array` extends `ConstArray` by adding assignment and indexing that returns element references), but it allows an `Array` to be passed as argument to a function that takes a `ConstArray`. For conciseness, this paper will focus on properties of the `Array` class. However, most of this discussion applies to `ConstArray` as well.

POOMA domain objects can be used to select a subset of an array. Rather than providing a special subarray class, POOMA makes further use of the engine concept: subscripting with a domain object creates a new `Array` object that has a different engine, a *view-engine*. View-engines reference a subset of the data owned by some other engine. For example, a brick-view engine, `Engine<Dim,T,BrickView>`, can be used to access any constant-stride, regular subset of points managed by a brick-engine.

The pairing of storage-engines and view-engines is handled via a traits class:

```
template <class Engine, class Domain> struct NewEngine { };
template <int Dim, class T>
struct NewEngine< Engine<Dim,T,Brick>, Range<Dim> >  {
  typedef Engine<Dim,T,BrickView> Type_t;
};
```

where `Range<Dim>` is a domain class that specifies a constant stride, rectangular subset of the points on which the brick-engine is defined. For example:

```
Array<1> a(10);                       // {a(0), a(1), ... a(9)}
Range<1> I(4,8,2);                    // {4,6,8}
Array<1,double,BrickView> b = a(I); // b(0) = a(4), ...
```

View-engine arrays will rarely be explicitly declared. However, these are constructed automatically in many array expressions, such as the following centered-difference:

```
Array<1> a(10), b(10);
Interval<1> I(1,8);
a(I) = b(I+1) - b(I-1);
```

where `Interval<Dim>` is a unit-stride version of `Range`. The last expression will create temporary view-arrays that will appear in the expression template object that is manufactured by PETE.

3 Expression Templates

Our goal is to write array expressions, such as

```
A = B + 2 * C;
```

and achieve the same efficiency as with explicit loops. Overloading the various operators to directly perform their operations cannot achieve this goal as evaluation will then happen in a pairwise fashion and involve multiple temporaries. Efficient evaluation requires the use of expression templates [3, 4, 5, 6].

Expression templates are a mechanism by which the parse tree for an expression is represented by a recursive template type. Both the type and an object of that type are manufactured by the overloaded operators as the expression is parsed. For instance, the expression `A = B + 2 * C` is represented by the parse tree shown in Fig. 1. In PETE, this parse tree would be encoded in an object of type

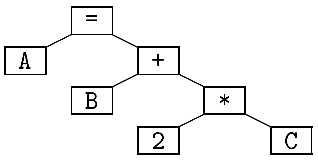

Fig. 1. Parse tree for the expression `A = B + 2 * C`.

```
Expression< TBTree<OpAssign, Array1,
            TBTree<OpAdd, Array2,
            TBTree<OpMultiply, Scalar<int>, Array3> > > >
```

where `TBTree<Op,E_l,E_r>` is a template class that stores an applicative template object, of type `Op`, and left and right subexpression, of types `E_l` and `E_r`. `Array1`, `Array2` and `Array3` are the types of `A`, `B`, and `C`. The `Expression` object can be used to efficiently evaluate the expression, as will be explained below.

4 PETE

PETE provides tools to add expression template capability to client classes. Generic techniques are used to adapt to a variety of class interfaces and to minimize the impact on client classes. Unlike other expression template implementations, PETE separates the construction of the expression object from the operations that can be performed on that object, allowing functors to be written that perform general purpose parse-tree traversal at compile time.

PETE is an adaptable expression template library similar to that described by Furnish [6]. PETE can be used in two modes. The method demonstrated here uses a combination of inheritance and templates, similar to the approach used in Ref. [6], to enable the client class to be interpreted as a terminal (leaf node) in the parse tree.[1] For example, consider a simple one-dimensional vector class, `Vec3`, that wraps a `double[3]` array:

[1] This template idiom was first discussed by Barton and Nackman in [7].

```
class Vec3 : public Expression<Vec3>  {
  double d[3];
  // . . .
};
```

PETE can also be used in a completely external manner, requiring no modifications to the client class. There are advantages and disadvantages to each approach, and the latter will be discussed briefly at the end of the section.

Unlike Furnish's `Indexable` base class, `Expression` makes no assumptions about the client's interface. Rather, `Expression` provides member functions that perform static up-casts to the daughter type:

```
template<class WrappedExpr>
class Expression {
public:
   typedef WrappedExpr Wrapped_t;
   Wrapped_t & peteUnwrap()
      { return *static_cast<Wrapped_t *>(this); }
   const Wrapped_t & peteUnwrap() const
      { return *static_cast<const Wrapped_t *>(this); }
};
```

Using these members, functions that operate on the terminals can directly apply the client's methods.

When using PETE in this mode, the client class must provide two typedefs, `Element_t` and `Expression_t`, and a member function:

```
const Expression_t &makeExpression() const;
```

`Element_t` is the element-type for the container. `Expression_t` and `make-Expression` allow the client class to specify an object of a different type to be used to generate the client's values. For example, some classes provide STL-like iterators rather than indexing. In this case, `Expression_t` would be the iterator type, `makeExpression` would construct and return an iterator object, and the iterator objects, not the client objects, would appear as terminals in the parse tree that PETE constructs. For clients that can efficiently produce their values directly (via indexing, internal cursor incrementing, etc.), `Expression_t` should be the client class itself.

`Vec3` can be indexed directly, so the following must be added to its public interface:

```
   typedef double Element_t;
   typedef Vec3 Expression_t;
   const Vec3 &makeExpression() const { return *this; }
```

These are all that are required for PETE to build expression objects from expressions involving `Vec3` objects. PETE's overloaded operators will recursively build an expression object, such as that described in Sect. 3, with `Vec3` objects at the leaves. A bit more work is require to evaluate the expression.

PETE operates on expressions using a template metaprogram that recursively walks the parse tree, applying a generic *tag-functor* at the terminals and a generic *tag-combiner* at the non-terminals. Tag-functors are specializations of the class:

```
template <class FTag, class Expr>
struct TagFunctor {
      typedef ???? Type_t;
      static Type_t apply(const Expr &e, const FTag &f);
};
```

This class must define a static member `apply` that takes the tag-object `f` and operates on the expression object `e` in the appropriate manner. It must also export the return type, `Type_t` (indicated by `????` here and below since C++ has no mechanism for declaring that a class will export a particular typedef).

At a minimum, the client class must provide a specialization of `TagFunctor` that returns the values of its elements. For example, in order to evaluate a `Vec3` object at index `i`, we define a tag indicating this purpose, and specialize `TagFunctor` to take the appropriate action:

```
struct EvalFunctor1 {
   int i_m;
   EvalFunctor1(int i) : i_m(i) {}
};

template <>
struct TagFunctor<EvalFunctor1,Vec3::Expression_t> {
   typedef Vec3::Element_t Type_t;
   static Type_t apply(const Vec3 &a, const EvalFunctor1 &f)
      { return a[f.i_m]; }
};
```

Note that the `EvalFunctor1` object carries the index with it.

The remaining task is to write `Vec3`'s assignment operator. The recipe for this is fairly straight-forward, but first the full traversal functionality will be explained as it has a number of uses.

In order to evaluate the expression, the traversal needs to combine the values returned by the tag-functor using the appropriate operators. This is accomplished with a special tag-combiner. For binary operators, tag-combiners are specializations of

```
template<class A, class B, class Op, class CTag>
struct TagCombine2 {
   typedef ???? Type_t;
   static Type_t combine(A a, B b, Op op, const CTag &c);
};
```

This class must define a static member `combine` that takes the tag-object `c` and the operator object `op`, and operates on the expression objects `a` and `b` in the appropriate manner, returning some result. It must also export the return type. Note that `A` and `B` may be reference types—it is up to the various `TagFunctor` specializations to define their return types appropriately if reference semantics are needed.

For the purposes of evaluating the expression, PETE uses an empty tag structure `OpCombineTag`, specializing `TagCombine2` as follows:

```
template<class A, class B, class Op>
struct TagCombine2<A,B,Op,OpCombineTag> {
   typedef typename BinaryReturn<A,B,Op>::Type_t Type_t;
   static Type_t combine(A a, B b, Op op, OpCombineTag)
      { return op(a,b); }
};
```

Here the applicative template object `op` is applied to the subexpression values and the result is returned. `OpCombineTag` serves only as a tag; the tag-object is not involved in the combine operation.

`BinaryReturn` is a traits class that uses C++ operator semantics and promotion rules to compute the type that results when the binary operator `op` is applied to the subexpressions `a` and `b`. This class could be implemented by enumerating all possible specializations, but this quickly becomes unwieldy. Instead, the type is computed via a somewhat complex set of template classes. While this is an important example of generic programming in PETE, the details are beyond the scope of this paper. We note, however, that the scheme is designed to be extensible by users.

Finally, we come to the traversal of the parse tree. This is handled by the `ForEach` template class, which has a familiar structure:

```
template<class Expr, class FTag, class CTag>
struct ForEach {
   typedef ???? Type_t;
   static Type_t
      apply(const Expr &e, const FTag &f, const CTag &c);
};
```

`ForEach` defines an `apply` method that operates on the expression with either `f` or `c`, depending on whether the expression is a terminal node or not. The default definition assumes that the expression is a terminal node, and thus applies the `TagFunctor`:

```
template<class Expr, class FTag, class CTag>
struct ForEach {
   typedef typename TagFunctor<FTag,Expr>::Type_t Type_t;
   static Type_t
      apply(const Expr &expr, const FTag &f, const CTag &)
          { return TagFunctor<FTag,Expr>::apply(expr, f); }
};
```

The recursion is performed by specializations of `ForEach` for the various types of non-terminal nodes. For example, binary expressions are handled by specializing on `TBTree` as follows:

```
template<class Op, class A, class B, class FTag, class CTag>
struct ForEach< TBTree<Op, A, B>, FTag, CTag >
{
   typedef typename ForEach<A, FTag, CTag>::Type_t TypeA_t;
   typedef typename ForEach<B, FTag, CTag>::Type_t TypeB_t;
   typedef typename
      TagCombine2<TypeA_t, TypeB_t, Op, CTag>::Type_t Type_t;

   static Type_t apply(const TBTree<Op, A, B> &expr,
                       const FTag &f, const CTag &c) {
      return TagCombine2<TypeA_t, TypeB_t, Op, CTag>::
         combine(
            ForEach<A, FTag, CTag>::apply(expr.left(),  f, c),
            ForEach<B, FTag, CTag>::apply(expr.right(), f, c),
            expr.value(),
            c );
   }
};
```

The first two typedefs compute the types that will be returned by application of `ForEach::apply` to the left and right subexpressions of the `TBTree`, and the third specifies the type that *this* `ForEach::apply` will return. The `apply` method recursively calls `ForEach::apply` on the left and right subexpressions. The results are then passed, along with the `TBTree`'s operator object (the "value" stored by the `TBTree` node) and the combine-tag object, to `TagCombine2::combine`. Finally, the result of this call is returned. Note that all of this occurs during inline template instantiation.

PETE also specializes `ForEach` for unary nodes, which are used to represent unary operators and element-wise function calls, and ternary nodes, which are used to represent where-expressions. These are quite similar to the above.

Note that `ForEach` does a *post-order* traversal of the parse tree, visiting a `TBTree`'s "value" *after* visiting its children. This is the appropriate traversal for expression evaluation. One can write similar functors to do more general traversals. For instance, a functor that prints a representation of the expression (without building that representation in a buffer) requires a more general traversal. Although PETE does not include more general traversal functors, they are not difficult to construct once PETE's `ForEach` functor is understood.

`ForEach::apply` could be written directly as a template function, as could the `combine` and `apply` methods of the tag-combiners and tag-functors. However, the `Type_t` traits that these classes export greatly simplify the rest of the code, so it seems more natural to put the implementation of the functor methods in the respective functor classes, providing template function wrappers where this simplifies the user interface.

Such a wrapper is provided for `ForEach::apply`. This is handy since template functions can deduce their template parameters from their argument types:

```
template<class Expr,class FTag,class CTag>
inline typename ForEach<Expr,FTag,CTag>::Type_t
forEachTag(const Expr &e, const FTag &f, const CTag &c) {
   return ForEach<Expr,FTag,CTag>::apply(e, f, c);
}
```

Now that the evaluation process has been explained, we can write the assignment operator for the `Vec3` class:

```
template <class Expr>
Vec3 &operator=(const Expression<Expr> &exp)
{
  Expr e = exp.peteUnwrap();

  d[0] = forEachTag(e, EvalFunctor1(0), OpCombineTag());
  d[1] = forEachTag(e, EvalFunctor1(1), OpCombineTag());
  d[2] = forEachTag(e, EvalFunctor1(2), OpCombineTag());

  return *this;
}
```

This function is almost trivial: `forEachTag` is called, passing it the expression, an `EvalFunctor1` object that propagates the index to the terminal nodes, and an `OpCombineTag` object that serves only to choose the `TagCombine2` specialization that applies the `TBTree`'s operator to the results returned from its children.

With the definition of the assignment operator, our `Vec3` class has all of the features necessary to use expression templates. The following code snippet will now compile and run:

```
Vec3 a, b, c;
b[0] = 10; b[1] = 3; b[2] = 2;
c = 1;
a = b + 2 * c;
```

Handling of scalars is provided by PETE, which defines a `Scalar<T>` template that is PETE-aware. The final expression will compile to the equivalent of:

```
d[0] = b[0] + 2 * c[0];
d[1] = b[1] + 2 * c[1];
d[2] = b[2] + 2 * c[2];
```

which is exactly the desired result.

If we were instead dealing with a class that provided an iterator interface, the example would be only slightly more complicated. As mentioned above, we would have to provide a `TagFunctor` specialization, say on `EvalFunctor0`, that dereferenced the iterator, and another, say on `Increment`, that incremented the iterator. The assignment operator would then look something like

```
template <class Expr>
Vec &operator=(const Expression<Expr> &exp) {
  Expr e = exp.peteUnwrap();
  iterator i = begin();
  while (i != end()) {
     *i++ = forEachTag(e, EvalFunctor0(), OpCombineTag());
     (void) forEachTag(e, Increment(), NullCombineTag());
   }
  return *this;
}
```

Here the first `forEachTag` returns the value of the right-hand side for the current iterator position and assigns it via the local iterator. The second `forEachTag` bumps the iterator. Its return value is ignored. Note that `apply` takes its `Expr` argument as a const reference. This is necessary since expression objects are often temporaries. Some tag-functor specializations, such as `Increment`, will have to cast away the const-ness in order to increment the iterators. Obviously this requires some care.

Evaluation is not the only use for `ForEach`. Another example is checking conformance of dynamic arrays. With the proper specializations, the code to do this would look something like

```
Expr e = exp.peteUnwrap();
int size = forEachTag(e, SizeTag(), AssertEqTag());
```

Here `TagFunctor<SizeTag,Expr>::apply` would return the size of each terminal, and `TagCombine2<A,B,Op,AssertEqTag>::combine` would assert that the values were equal, and if they were, it would return the value.[2] In POOMA, this idea is extended to checking domain conformance, returning the common domain as a result.

`ForEach` can also be used to construct a new expression tree from the original one. POOMA uses this capability to build expression trees whose terminal nodes contain views of the terminal arrays in the original tree.

As was mentioned earlier, PETE can be used without modifying the client class. Instead, the required functionality is put in an external traits class, `MakeExpression`. Unfortunately the assignment operator cannot be handled in this manner since C++ requires that it be a member function. As a result, a completely external implementation can be only be achieved if the user is willing to forgo assignment for an external `assign` function. The biggest disadvantage of the external approach is that clients must define the appropriate overloaded operators to build the expression elements. PETE includes an example PERL script that is used by POOMA for this purpose. While this is more complicated to implement than the inheritance approach, it does avoid certain problems due to subtleties of the C++ template matching algorithm. POOMA employs a mixed approach, using `MakeExpression`, generated operators, and an overloaded assignment operator.

[2] Life is actually a bit more complex due to interactions with `Scalar<T>`.

5 Expression-Engine

As we have seen, an expression object can be used to generate an optimal set of loops for evaluation, and it can be manipulated by PETE functors to query its properties, build new trees, etc. However, an expression cannot be passed to a function expecting an `Array`, which is a desirable capability. Implicit conversion from expressions to arrays will not work as these are not performed when matching templates. Even if they were, it would be undesirable to create a temporary to hold the result. POOMA's engine architecture provides an elegant solution to this problem. An engine object can be constructed that contains an expression object and uses functors to index the expression. POOMA implements this as:

```
template <int Dim, class T, class Expr>
class Engine<Dim,T,ExpressionTag<Expr> > ;
```

PETE's template machinery is used to efficiently calculate values when the array is indexed, avoiding unnecessary evaluation.

As a result, any template function that takes a `ConstArray` having an arbitrary engine type can be called with an array expression. For example, if we have the function

```
template<int Dim, class T, class ETag>
T trace(const ConstArray<Dim,T,ETag> &a) {
  T tr = 0;
  for (int i = 0; i < a.length(0); ++i) { tr += a(i,i); }
  return tr;
}
```

Then `trace(B+2*C)` sums the diagonal components of `B+2*C` without computing any of the off-diagonal values.

Adding an `Array` interface to an expression also simplifies the evaluation code. POOMA exploits the separation of expression construction and evaluation by deferring the evaluation to a separate set of template classes, called *evaluators*. The `Array` assignment operator constructs a new array whose engine contains the expression `*this = rhs`; i.e., an expression with the element-wise assignment operator at the root of the tree. This new array is then handed off to the evaluator. A detailed discussion of evaluators is beyond the scope of this paper. However, the simplest evaluators ultimately expand to the following loop (for a two-dimensional array):

```
for (int i = 0; i < n0; ++i)
  for (int j = 0; j < n1; ++j)
    expr(i,j);
```

where `expr` is the `Array` object passed to the evaluator.

6 Summary

This paper has presented generic techniques used in POOMA and PETE to achieve flexibility without sacrificing efficiency. The POOMA array-engine abstraction separates the representation of an array from its interface. This greatly simplifies the development of new array types as one need only build the appropriate engine. The `Array` class provides the interface, including the interaction with expression templates.

POOMA's expression templates are packaged as a separate, reusable package, PETE, that makes extensive use of traits, template metaprograms, and other generic techniques to maintain container-independence and to simplify type computations. PETE can be used via inheritance, in which case one makes minor additions to the client's interface, and writes an assignment operator that takes an expression object and uses the `ForEach` functor to do the evaluation. PETE can also be used in a completely external mode, or in a mixture of the two.

POOMA further exploits the separation of the expression and its evaluation. Arrays can have expression-engines, allowing expressions to be passed to functions expecting `ConstArray` objects, with expression evaluation occurring when the array is indexed. Furthermore, arrays delegate evaluation of expressions to separate evaluator objects, allowing specialization of evaluators for certain execution environments and certain types of terminal engines.

Acknowledgements

We would like to thank Geoffrey Furnish and Todd Veldhuizen for many valuable discussions. The research described here was performed under the auspices of the U. S. Department of Energy by Los Alamos National Laboratory under contract No. W-7405-Eng-36.

References

[1] John Reynders et al. POOMA: A framework for scientific simulations on parallel architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 553–594. MIT Press, 1996.

[2] William Humphrey, Steve Karmesin, Federico Bassetti, and John Reynders. Optimization of data-parallel field expressions in the POOMA framework. In *ISCOPE '97*, December 1997. Marina del Rey, CA.

[3] Scott Haney. Is C++ fast enough for scientific computing? *Computers in Physics*, page 690, November 1994.

[4] Todd Veldhuizen. Expression templates. *C++ Report*, pages 26–31, June 1995.

[5] Scott W. Haney. Beating the abstraction penalty in C++ using expression templates. *Computers in Physics*, page 552, November 1996.

[6] Geoffrey Furnish. Disambiguated glommable expression templates. *Computers in Physics*, page 263, May 1997.

[7] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.